CHAPTER

5

Test

Continuously Championing Quality

The volume and velocity of software innovation afforded by DevOps is perhaps the biggest driver of IT's shift to this new method of delivery. But when organizations neglect quality in the head-long rush to DevOps glory, the glass can only ever be half empty.

The business-technology landscape is littered with many examples of what happens when software speed has been pursued at the expense of quality. Perhaps the most extreme is Knight Trading, where a software update accessed outdated code (8 years old) that made more than $440 million in bad trades in less than 30 minutes.[1],[2]

DevOps principles and practices are therefore not only intended to improve the tempo of software releases but also increase quality—and as with delivery, this must happen continuously!

However, with DevOps' focus on an automating the software pipeline, it's clear that traditional methods for ensuring quality must now be questioned and reviewed. Separate teams working in silos, working with centralized polluted test data, and performing manual tasks late in the software development cycle is no way to sustain quality.

[1] http://www.bloomberg.com/news/articles/2012-08-02/knight-shows-how-to-lose-440-million-in-30-minutes
[2] https://www.sec.gov/litigation/admin/2013/34-70694.pdf

A. Ravichandran et al., *DevOps for Digital Leaders*, DOI 10.1007/978-1-4842-1842-6_5

Progressive businesses understand implicitly the connection between speed and quality; increasing the cadence of releases, yes, but championing quality through the application of advanced DevOps automation. One illustrative example is AutoTrader.com, the online marketplace for car buyers and sellers.

Case Study: AutoTrader.com

Every month more than 18 million people use AutoTrader.com to search for a used car. The site does more than host advertising for sellers; it also helps people research and compare cars and trade in their old vehicle.[3]

Based in Atlanta, Georgia, AutoTrader's goal is to make exchanging vehicles simpler and more secure than ever before, while maximizing value for private and trade buyers and sellers.

When AutoTrader's services were mainly available through a browser, it saw itself as a media company. The emergence of smart mobile devices and the application economy has changed all that.

"Right now we're a technology company. The technology aspect is very important to AutoTrader as a business," reveals Adam Mills, Senior Manager of Application Development at the company.

Ten years ago AutoTrader released just four web services updates a year; today it expects to release one almost weekly. As Mills explains, "We have to keep up with changes to current operating systems and devices as well as evolve our own offerings. Customers expect us to deliver great new functionality in weeks rather than months."

Customers also expect the same excellent experience whether they are accessing AutoTrader via an app or a browser on a mobile, desktop, or laptop.

In a highly competitive market, this excellent experience is a key differentiator for AutoTrader. "Our app has to be the best," explains Mills. "All it takes is a couple of bad customer experiences then everybody's talking about it on Twitter and we lose market share."

As the functionality of AutoTrader.com and the number and variety of devices it supported grew, software testing became complex, costly, and time-consuming. "We had to set up huge emulation environments, buy the licenses, and ensure all the services were talking to each other," recalls Mills. "But because there were so many interdependencies, we couldn't complete all the different tests in the same timeframe. We then had to find all the physical devices, plug them in, and test our code on every one."

---

[3]Full Story: http://www.ca.com/content/dam/ca/us/files/case-studies/autotrader-avoids-300000-in-testing-costs-with-ca-service-virtualization.pdf

With plans already underway to adopt a DevOps approach to bring together its disparate development teams, AutoTrader realized that virtualizing different services would enable even greater unity.

AutoTrader searched the market for a service virtualization solution that supported the DevOps approach, inviting a select number of vendors to demonstrate the solutions.

After implementing a solution in six weeks, AutoTrader began using the solution to simulate apps behaving normally and performance issues. "Teams can test how resilient their services are and answer those key 'what if' questions, like 'what happens if the database crashes?'" comments Mills.

Mills envisages that soon the last human interaction with a piece of code will be when a developer checks it in. Test, build, and deployment will be automated, reducing processes that previously took weeks to just minutes.

AutoTrader has been able to accelerate testing, while improving quality and freeing up resources. As Mills confirms, "The solution means we can complete testing in hours rather than weeks. Previously we would have needed hundreds of testers to check performance on every device, but now we can test all devices automatically while our team focuses on higher value activities."

The time taken to set up a new testing environment has also been cut from two weeks to two days, with costs dramatically reduced. AutoTrader.com has been able to:

- Cut integration time from three days to three hours
- Save and average of 567 man-hours—or 2.5 people—per release
- Avoid $300,000 in test hardware and software costs
- Decrease software defects by 25 percent

As Mills concludes, "By getting new releases and services out the door quickly, we can provide a better experience to millions of car buyers and sellers and continue to differentiate in a competitive market."

Testing Times

Apart from illustrating the importance of software quality, the AutoTrader story shows that this doesn't have to slow things down. As the forward-thinking Mills suggests, automation will be key for testing to become established within both continuous integration and continuous delivery processes.

Testing is essential to DevOps because it brings the discipline smack-bang into the development processes and avoids the problems (e.g., release delays and quality issues) created by leaving QA as a gate or rubber-stamp function only performed at the very end of the cycle.

This isn't to say that the role of tester will be subsumed with development, but the discipline will change. Rather than providing a transactional service to developers (e.g., executing tests and handballing the bad news), the focus of testing will shift toward a more consultative role that will help developers learn how to write better tests and improve their approaches to scanning for quality. Developers aren't necessarily hard-wired to look for quality issues, and even though the vast majority do care about quality code, they are still going to miss issues and opportunities for improvement.

Tip To establish testing expertise DevOps style, leaders should consider positioning their teams in a way that can add the most value across the software development lifecycle. This may involve embedding specialists within agile product teams or even creating a center of excellence.

As advanced automation becomes more pervasive, QA and testing professionals will need to become better skilled at fully leveraging it. This involves providing a comprehensive and elevated test discipline rather than just executing a series of day-to-day tasks.

Some new skills include:

- *Thinking beyond pass or fail*—Helping the business understand what the customer actually experiences and how that can be best simulated during testing. Essentially supplying the right data and real-world conditions needed to better support and enhance a quality experience.
- *Intimate understanding*—With the complexity surrounding applications today, QA, and testing staff need to become far more proficient at understanding all the intricacies. At a minimum, this means visualizing all dependencies and being able to remove constraints.
- *Assurance to analytics*—QA has traditionally been focused on documenting defects and reporting back to development. This must shift toward collecting and aggregating data from a broad range of automated tests to determine the actual cause of defects and where more rigorous testing is needed.
- *Early and thorough testing*—With agile increasing the volume of user stories, it makes perfect sense to incorporate testing into the acceptance criteria. At this early stage any progression into the sprint should be dependent on reviews involving QA, but also security and IT operations too.

■ **Note** With agile and DevOps, quality is baked into the SDLC, not bolted on at the end. This requires establishing ownership at a cross-functional level, not devolving to one team. Automation to support this goal should be available to all stakeholders, not just QA/testing teams.

- *Mentorship over conflict*—Rather than constantly being called in to address fragile developer-written tests, DevOps focused QA will work closely with their coding colleagues to continuously improve testing resilience.
- *Ambiguity to clarity*—Vague requirements stored in multiple formats leads to defective software and a sub-optimal customer experience. Teams should seek out methods to map changing requirements to visual models and eliminate ambiguous requirements and the costly defects they create.
- *Quality over quantity*—Having many redundant, duplicate tests guarantees nothing but cost overruns and delays. QA and testing teams should consider advanced automation methods that generate the smallest number of test cases needed for 100 percent functional coverage—all linked to the right data and expected results.

Agile Testing Trifecta

A key goal of DevOps should be making testing an accelerator, not an obstacle to fast application delivery with the highest levels of quality. To support this, more advanced testing tools are needed, equipping QA and testing teams with three essential capabilities needed to support agile and continuous delivery methods. This "testing trifecta," as illustrated in Figure 5-1, includes:

- *Test automation* to create test cases right from requirements
- Generating *synthetic test data* to be used on demand
- *Test constraint removal* by virtualizing every environment that needs to be accessed

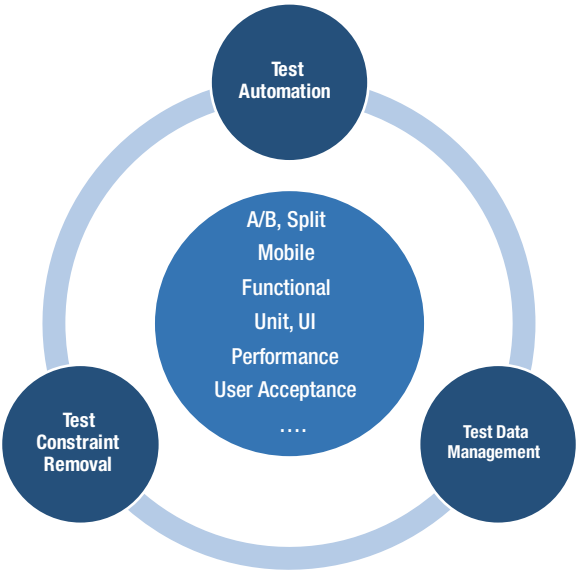

Figure 5-1. Testing trifecta for agile and DevOps

Test Automation

Today, nearly every company is in the software business. Although an organization may sell a tangible product, their use of software to streamline and enhance the customer experience means they must place higher importance on quality application delivery. Frequently, applications that are rushed through the development cycle without adequate testing often encounter costly defects that impact the customer relationship.

Just like building a house, the foundation is key to successful software construction. If the foundation has issues, there is a high likelihood for expensive delays further into the process. Using the right development tools at the onset will help ensure that the software foundation is properly defined, constructed, and tested while keeping quality and end user goals top of mind.

Incomplete Requirements Equals Faulty Software

Many quality problems eventuate during the requirements design phase. This is because software requirements are typically ambiguous, incomplete, and stored in many different formats by numerous people within the organization. Test cases are then manually defined from incomplete requirements and thus the stage is set for foundational cracks to appear even before the application has been built.

Further, the manual definition of test cases is a slow and unsystematic process that leads to perhaps 10-20 percent functional test coverage. Testers end up testing the same features over and over again without knowing for certain the results. As a consequence, defects are detected later much in the development cycle, leading to costly rework.

An Automated and Agile Approach

If testing is going to keep pace with continuous delivery goals, it needs to become much more automated and agile. Adopting a requirements-driven (or customer centric) approach is the first step and may require software solutions to force the change. With the advanced tools, testers can generate the right test cases needed for maximum coverage. Test assets can be derived directly from the design and updated automatically to reflect changing user needs.

Tools in this category allow user stories to be imported and modeled as an active flowchart (see Figure 5-2).

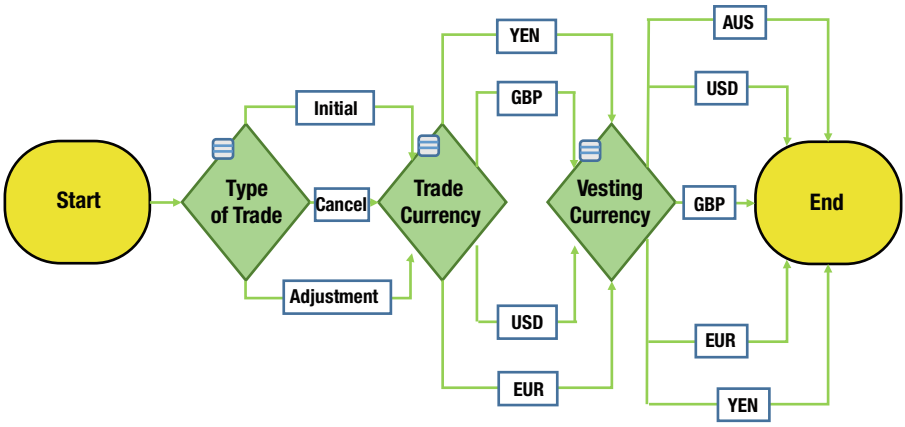

Figure 5-2. Agile requirements design allows user stories to be verified with end users

Active flowcharting helps eliminate requirements ambiguity and reduce defects early in the design phase. This class of tool will also generate the smallest set of automated tests needed for maximum coverage. Importantly, and to support the drive to testing as a discipline becoming much more proactive, these tools also help testing teams know which features should receive the most rigorous testing based on analytics and metrics gathering capabilities.

Achieving Complete Test Coverage

As applications become more complex and distributed, business logic is no longer found only in the user interface (UI) and the database (as with client/server models), but extends across multiple tiers and technologies. This becomes further complicated when applications consume underlying services from cloud providers or third-parties, or use highly interactive presentation layer technologies.

Organizations are also implementing more agile development methods from distributed teams, yet the use of shareable, reusable test assets between these teams is limited or non-existent. Traditional tools designed for more linear style waterfall development are often employed, but lack extensibility, only supporting the needs of one group. For example, code-based unit testing tools for developers that are unusable by QA and functional user interface (UI); failing to translate errors into repeatable defect identification needed by developers to catch bugs earlier.

This requires a much higher degree of test automation and collaboration among stakeholders. As testing efficiency and effectiveness become paramount, a new continuous testing model supported by advanced automation technologies should be the goal. Only through this coordinated approach can organizations build the scale needed to meet future demands.

Meeting these goals can only be ensured when every layer of the application and the complex interactions between components is automatically tested and verified throughout the software lifecycle. This involves providing complete test coverage with the ability to invoke and verify the behavior of each component, singularly or as an end-to-end service. Solutions in this class must therefore provide industry-leading standards support, with native integration to J2EE servers, integration suites, and ESBs. To help strengthen the DevOps toolchain, solutions will also integrate popular open source tools (e.g., Selenium Builder for UI testing), thereby enabling end-to-end testing from user interface all the way to back-end systems.

Case in Point: Mobile Testing

True extensibility means one tool coordinating and running functional tests, test UIs, and APIs on multiple mobile devices under various conditions. Traditional approaches to testing fall down in the mobile world because it's no longer sufficient to just to test the "function" of the application. Code needs to be tested using the same conditions that the app will run under when in the hands of a user, with experience-based metrics and test reports reviewable by both the user acceptance teams and development to further improve quality and expedite defect resolution.

To support the goal of ensuring high-quality during continuous integration (critical for mobile apps where changes updates occur more frequently), such solutions should provide unattended automation coverage. This involves executing tests against real mobile devices connected locally or from the cloud (see Figure 5-3) immediately code is committed.

Virtualized services described in more detail later in this chapter address the common mobile testing challenge of testers needing access to dependent systems for end-to-end analysis. Tests should also allow for different profiles simulating network conditions, location, background applications, and device orientation. This way teams can report and benchmark the user experience of different personas at various points in the app workflow.

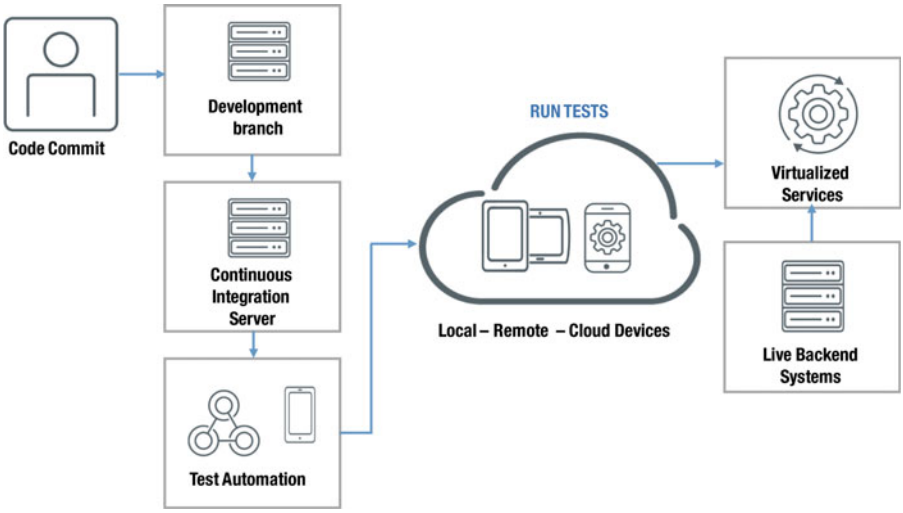

Figure 5-3. Automated mobile tests on smartphones, using multiple OS versions on multiple carrier networks and in different locations worldwide

■ **Tip** When mobile apps are in full production, consider using app experience analytics tools for continued insight into both usage and performance. Results can be valuable for determining where functional and performance improvements are needed.

Test Data Management

The second part of the testing trifecta and an area rife for improvement in the software development lifecycle is in test data management. Every tester needs quality test data and quickly. The challenge is getting the right data to

match their tests when they need it. As companies have improved their development processes, moving from Waterfall to agile, testing has lagged behind. Again, manual processes cannot keep pace with a company's test data need, with companies relying heavily on teams of people constantly creating and maintaining test data.

Another major challenge when managing test data is ensuring compliance with legal and regulatory requirements. Many organizations apply the necessary rigor when protecting personal and sensitive customer information in production, but neglect to consider the implications when working with data in non-production environments. In the event of non-compliance, this can mean significant consequences, not the least brand reputation, but also financial loss due to fines and penalties.

Many industry-specific regulations come with their own unique sets of test data challenges, and some introduce new complexities. Take the General Data Protection Regulation (GDPR) for example. The GDPR is designed to protect the rights of European Union (EU) citizens where the processing of their personal data is concerned.

Although many companies will have already adopted privacy processes and procedures consistent with the directive, the GDPR contains a number of new protections for EU data subjects and threatens significant fines and penalties for non-compliance (up to 4 percent of annual global turnover or 20m euros, whichever is greater) once it comes into force in May 2018.

GDPR introduces many new obligations in areas such as data anonymization, breach notification, and trans-border data transfers, to name just a few. Many have implications for test data management. One example is the "right to erase," where individuals may notify businesses processing their data what they may or may not use that data for, including testing.

Complying with obligations like this carries a huge overhead. If customers state they don't want their data used (even it is masked), then testers will need to acquire subsets of data and apply filtering rules. They will also need to ensure their methods can track every record not approved for testing and be fully auditable.

Many businesses might pursue programmatic solutions to these test data problems, but this only increases the development burden and potentially introduces additional fragility. One alternative, of course, is to use more modern synthetic test data generation to avoid these problems completely.

Facets of a Gold Standard Solution

To address the complex issues involved with the acquisition of quality data and regulatory compliance, modern test data management solutions will provide:

- *Synthetic test data generation*—Synthetic data contains all of the characteristics of production but none of the sensitive content. This ensures teams are provisioned with secure, realistic data that maintains referential integrity as part of a move toward a "Live Data Exclusion" model for testing. In addition to addressing compliance issues, and as illustrated in Table 5-1, synthetic test data generation can address other constraints.

Table 5-1. Removing Constraints with Synthetic Test Data

Constraint	Resolution
Regulations and compliance	**Lower risk as data is generated**
Data functional coverage	Measure and get 100 percent coverage
	Capacity to identify data "holes" by comparison between environments or directly identify data from test cases
Test database size	**Only stores most efficient set of test data**
Provisioning delays	Provisioning in minutes, on-demand, through a web portal
	Capacity to book data for each tester

- *Scalable end-to-end platform*—Tools should clone subsets of data into target environments and be capable of securing millions of data rows in minutes using automated data profiling and advanced masking engines.
- *Test data allocation*—Tools must facilitate automated data discovery for testers to receive exact datasets, linked to their test cases.
- *Test data warehouse*—The ability to store pools of test data as reusable assets in a central repository and test multiple versions and releases in parallel.

The following checklist can also be useful in assessing the efficacy of test data management solutions:

- Provides a standard set of data to test
- Is "*production-like*"

- Covers all possible tests that need to be run, including future and negative scenarios
- Contains *just enough* data to test repeatedly
- Is up-to-date, while also containing and supporting all previous data
- Contains absolutely *no* sensitive data

Combining with Test Automation

The *test automation* methods described in the first part of this section, especially the ability to create test cases right from requirements, are powerful capabilities in their own right. However, combine them with test data management and testers can move beyond just executing tests to proactively driving quality improvements.

By way of example, consider a two-way integration between test data management and agile requirements definition. Here, test matching functionality should be available to locate or create the data needed to execute the optimized test that has been built straight from requirements. The test data itself would be stored in a central test data warehouse where it can be provisioned on demand and used in parallel with development efforts.

Through dynamic building, testers can request the data they need based on specific criteria and receive it in minutes from a self-service web portal. The provisioned data is cloned and version controls are applied to update data to immediately reflect any changes in requirements.

Using the integrated approach, teams benefit in many ways:

- Distributed test teams can work with multiple application versions with matching test data
- Automatically locate or create test data based on specific testing needs
- Test for outliers, unexpected results, and negative scenarios
- Significantly reduce the time and resources required to provision test data
- Generate synthetic data (data from scratch) without the need to mask production data
- Create test data quickly for use in service virtualization to speed testing and increase quality; feeding data directly to service virtualization engines and linking test data with virtual end-points

Implementing a test data management strategy is crucial to realizing the goal of continuous application delivery. By making test data accessible during requirements design, teams can streamline and eliminate the bottlenecks associated with test case creation and locating the right test data.

Test Constraint Removal

There is a fundamental shift in the way enterprises build applications today. In the early days of mainframe and client/server applications, you had a much more limited scope of applications—all of the components from the database to the UI could be under one development and testing team's control.

After the Dot-Com days of the early 2000s, a new style of composite applications arose. The new approach to developing software, including agile, created two new challenges for organizations

- Constraints created by the highly parallel development efforts
- Dependencies on consistent behavior of the components in the system

These complications increased the complexity and cost of developing and maintaining composite applications.

Applications today are the result of many decades of "building systems on top of systems," which creates huge chains of dependencies. These complex architectures mean software development is more difficult, more costly, and more complex than ever before.

Many large organizations now find that many of systems they depend on such as mainframes, databases, and external services are constrained and not accessible by developers and testers when they are most needed.

For instance, a needed mainframe may be off-limits, a system of record could have bad data, or a third-party service may still be under development. Attempts to reproduce these environments—by manually coding stubs and managing test data—are costly and inconsistent.

One customer with constraint issues put it this way, "I can't do anything until I have everything… and I never have everything!"

In a recent Voke Market Snapshot Report on Service Virtualization (January 2015), over 500 companies validated that constraints are a major hurdle to innovation in the software development lifecycle.[4] The report mentions that:

- 80 percent of teams experience delays in development due to constraints everywhere across the SDLC

[4] https://www.ca.com/au/collateral/industry-analyst-report/voke-market-snapshot-report-service-virtualization-iar.register.html

- 56 percent of critical dependencies are unavailable when development and test need them
- 70 percent of teams face prohibitive restrictions (delays, time, and fees) when needing to access third-party systems

Service virtualization solutions can solve these constraint issues by capturing and modeling dependent systems. As virtual versions of the real thing, these services simulate the constrained components in any environment, providing low-cost, 24/7 available models.

When developers and testers use service virtualization, the services behave and perform similar to the real thing, but without the underlying hardware and software complexity of a physical system. Development and testing continue just as they always have, but less constrained, and without contention between teams for environments, labs, test data, and so on.

Although service virtualization solves many different development problems, four common ones are seen repeatedly:

- "Shift left"—Enabling parallel software development, testing, and validation for faster time-to-value with earlier defect resolution (see Figure 5-3)
- Infrastructure availability—Eliminating much of the concurrent demand for environments and hardware that agile development creates
- Performance readiness or solving the challenging problems of properly evaluating the scalability of applications
- Scenario and data management—Often eliminating the need for complex test data management, system setup, and other complexities

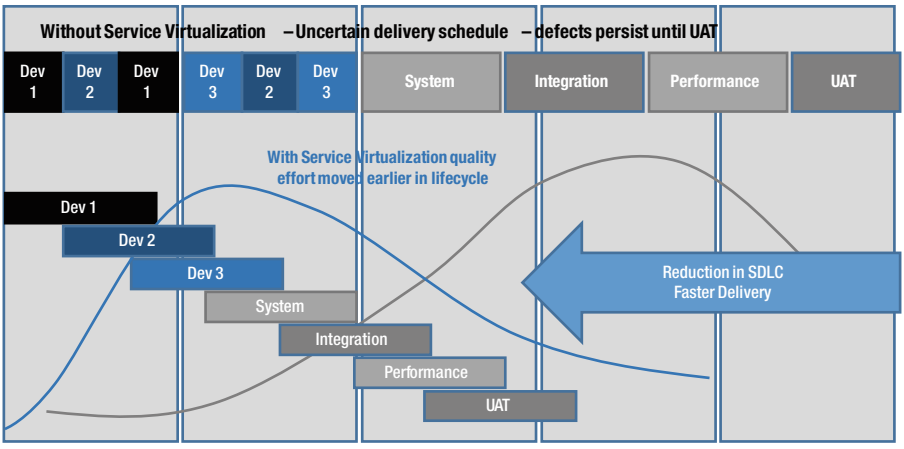

Figure 5-3. "Shift-left" testing with service virtualization

One common problem that service virtualization solves is in the area of integration. Customers buy companies, they provide service to third parties, or they are updating applications for functionality, compliance, or architecture. Each of these challenges presents an opportunity for service virtualization to improve the software development process. Integration teams and customers have the resources they need for software development and testing, without the added expense of acquiring additional hardware and software.

When time-to-market matters, service virtualization offers an excellent opportunity to shorten development lifecycles. Service virtualization reduces the constraints of software development, allowing more teams to effectively work in parallel, without underlying dependencies. Typically, service virtualization users experience a 25-50 percent reduction in release times.

A huge opportunity for service virtualization is in the area of performance engineering. Creating a lab capable of handling and testing to production capacity loads is difficult and resource intensive. Furthermore, ready access to systems such as mainframes and transaction servers may be impossible. All this makes performance testing expensive, unreliable, and inconsistent. Some teams might have a small window for testing, while others will have to wait for an entire application architecture to be assembled before any testing can commence. But by using service virtualization, teams can performance test each individual component, identifying many performance problems earlier in the lifecycle, and reduce, even eliminate, the amount of final performance testing needed in a production-like lab.

Using traditional Waterfall methodologies for developing software, much of the activity of development and testing of the application happens in a series of steps, one after another. But by eliminating constraints common in typical software development practices, service virtualization enables much of the SDLC to operate in parallel and the steps within it become less time consuming.

Using service virtualization, developers can have their own private environments for coding, directly from the laptop. They don't share environments and don't need to wait for other developers to finish their work.

With service virtualization, much of the testing at a component level can "shift left," or be moved earlier in the SDLC. Because each component can be tested individually (instead of waiting for a complete assembly), unit and regression testing happens sooner and is more complete, and defects are identified long before integration or user acceptance testing. Finding defects earlier means developers fix issues at the point in time they incur the lowest cost. This avoids defects leaking into later stages or even into production and become harder to resolve because developers have been moved onto other projects.

As teams increase service virtualization maturity, regression and individual component testing become increasingly automated. Now validation as early as code check-in is possible, making defect detection a consistent and repeatable process. Again this is possible because service virtualization allows component level testing in isolation, without underlying dependencies.

Once automation is implemented, you can easily make it a continuous process. Using this approach, any change breaking interfaces, contracts, or use patterns are easily detected before the code disrupts other services or applications.

The deployment of service virtualization at one large bank solved two critical challenges. First, by eliminating system dependencies, testing began far earlier in the development cycle. Defects in code no longer lurked until UAT, but were found much earlier.

Additionally, the bank's formally serial processes were set in parallel, dramatically reducing release times.

Using virtual services reduces the demand for physical hardware and test labs. This approach is distinctly different, and complementary to, hardware virtualization. With service virtualization, you virtualizes services and business functionality instead of hardware.

When demands for hardware decrease, so do costs. The challenges and costs of provisioning labs and equipment, software and configurations disappear. The physical hardware demand decreases dramatically, freeing budgets for application and business investments instead of capital assets. Demand for data center rack space, power, and storage also decrease.

In performance testing, service virtualization helps customers reduce cost and increase quality and flexibility in several ways. Customers can load test at the component level. Instead of waiting until the application is complete, components are tested for volume and capacity independently, locating bottlenecks and issues early.

Production-only systems such as master databases, mainframes, and third-party systems not normally available for load testing are virtualized, creating an always-ready, highly scalable virtual back-end immune to traditional load testing constraints. This benefits users both in convenience and by reducing the cost associated with replicating expensive back-end systems.

Customers may also face third-party access or software license fees. Service virtualization eliminates the need for highly scalable versions of these systems by virtualizing their behavior. For organizations selling services, virtualized versions of their entire platform are made available to customers in virtual form. Validating against a virtual back-end ensures production readiness without the complexities of full production-style dev/test implementations.

Using service virtualization addresses many thorny issues associated with test data management. For example, organizations struggled to set up just the right scenarios, only to "burn" them with a test cycle. Or, find it difficult to construct test scenarios for edge conditions and business logic. In such cases, it often becomes more expensive to set up the test harness than do the test!

Virtualizing behaviors such as edge conditions, negative test scenarios, and error handling are easily configured in the behavior of the virtual service and are never "burned" since the virtual service is simply playing back behavior responses.

With service virtualization, "test data" and scenarios are easily versioned and changed for each new requirement. In addition, when two test cycles or teams have differing needs for test data, they will not collide in the test lab.

Summary

With the advent of agile development, testing as a discipline is changing radically. Using the approaches described in this chapter, testing can move beyond being a separate siloed function employed at the end of cycles, to becoming a more proactive, continuous, and analytical discipline that firmly establishes quality, whatever the pace of the delivery.

While achieving this goal may require changes in mindset and organizational structure, what's indisputable is the need to adopt a comprehensive testing approach to address end-to-end automation needs, manage test data, and remove all constraints.

In the next chapter, we'll examine the software releases strategies organizations should consider as they move to a more continuous method of delivery.